Handcrafted Inversions Made Operational on Operational Semantics

Jean-François Monin, Xiaomu Shi

Handcrafted Inversions Made Operational on Operational Semantics

Jean-François Monin[1,2] and Xiaomu Shi[1]

[1] Université de Grenoble 1 - VERIMAG
[2] CNRS - LIAMA

Abstract. When reasoning on formulas involving large-size inductively defined relations, such as the semantics of a real programming language, many steps require the inversion of a hypothesis. The built-in "inversion" tactic of Coq can then be used, but it suffers from severe controllability, maintenance and efficiency issues, which makes it unusable in practice in large applications.

To circumvent this issue, we propose a proof technique based on the combination of an antidiagonal argument and the impredicative encoding of inductive data-structures. We can then encode suitable helper tactics in LTac, yielding scripts which are much shorter (as well as corresponding proof terms) and, more importantly, much more robust against changes in version changes in the background software. This is illustrated on correctness proofs of non-trivial C programs according to the operational semantics of C defined in CompCert.

1 Introduction

The work described here is motivated by an experiment reported in [3,14], called SimSoc-Cert (a certified simulator of Systems on Chips) where we develop proofs of C programs using the operational semantics of a large subset of the C language as defined in the CompCert project [6]. An important characteristic of our framework is the large complexity of the specification, driving us to use powerful features such as higher-order functions, dependent types, modules, not only for convenience, but in order to keep the specification as readable and reusable as possible. Still, its size is rather large by force, since it includes the behavior of several commercial processors (currently: ARM and SH4). In such a framework, there is little hope for full automation. Proofs are performed by alternating clues given by the human user and tedious steps that are expected to be automated. Though all this is well-known, the situation can become very tricky when automated steps produce goals with many new variables and hypotheses in the environment. In an interactive setting, their names can be refered to later in the script. This issue cannot be overlooked, despite the lack of a nice theory on massive names management – up to our knowledge. And it actually occurs with SimSoc-Cert, because proofs rely heavily on *inversion* steps on hypotheses relating memory states of the program, according to a large inductive transition relation which is the heart of the operational semantics of C defined in CompCert.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 338–353, 2013.

In a few words, an inversion is a kind of forward reasoning step, which allows us to extract all useful information contained in a hypothesis. It is nothing but a case analysis on a carefully prepared goal (more detail to come in Section 2). The practical need for automating inversion has been identified many years ago and most proof assistants (Isabelle, Coq, Matita,...) provide an appropriate mechanism. The first implementations for Coq and LEGO are analyzed and explained in [5] for Coq and [7] for LEGO. Since then, the main tool available to the Coq user is a tactic called `inversion` which, basically performs a case analysis over a given hypothesis according to its specific arguments, removes absurd cases, introduces relevant premises in the environment and performs suitable substitutions in the whole goal. This tactic works remarkably well, though it fails in rare intricate cases, as reported in mailing lists (see also Section 3.5). An additional approach called BasicElim was proposed in [8]. It is implemented in Matita [13], for instance. BasicElim is available in Coq as well.

However, the price to pay for the generality of `inversion` and BasicElim is a high complexity of underlying proof-terms. Does it reflect an unnecessarily complex formalization of a (at first sight) rather simple idea? A practical consequence is that unpleasantly heavy proof terms can unexpectedly occur in functions defined in interactive mode. For developments which make an intensive use of inversion, such as SimSoc-Cert, the evaluation of scripts is painfully slowed down.

However, the abovementioned issue on name management turns out to be still much more important: hardly controlled names are introduced in the environment. This would not be an issue if we don't see them, e.g., if the generated goals can be automatically discharged. But this is hopeless when dealing with complex specifications, as in our case. In general, the sequel of the script refers to generated hypotheses. Typically, introduced hypotheses could be inverted again, and so on. This poses a very serious problem of robustness: updating the inductive relation or even minor modifications in another part of the development may result in a complete renaming inside a proof script, which has then to be debugged line by line. In the previous stage of our work reported in [14], we could perform a proof on a single instruction of the ARM processor. So in theory, everything was solved. However, the number of inversion steps was so large this proof could not survive the various updates of Coq and CompCert.

The available version of `inversion` where explicit names can be given in the script (`inversion... as`) is better for robustness, but too heavy for our needs: each inversion would require the introduction of many (often more than ten) additional names. BasicElim raises similar issues, though its behaviour is more regular.

In order to get scripts which are both robust and much shorter, we want to provide programmable inversion tactics, requiring only a few explicit names. To this effect, we propose a handcrafted approach to inversion. The initial idea for this inversion was exposed in [9] (and is recalled here in Section 3.2) but, in order to be general enough, it had to be revisited with inspiration coming from the impredicative encoding of inductive datatypes.

The concrete setting considered here is the Coq proof assistant, but the technique can be adapted to any proof assistant based on the Calculus of Inductive constructions or a similar type theory, such as LEGO or Matita.

The rest of the paper is organized as follows. Section 2 recalls the basics on inversion. Section 3 explains our technique for performing inversions. Section 4 contains a summary of the application to SimSoC-cert. We conclude in Section 5 with a comment on our achievements and some perspectives.

2 Inversion

Type-theoretic settings such as Coq [15,2,4] offer two elementary ways of constructing new objects: functions and inductive types[1]. For instance, even Peano natural numbers can be inductively characterized by the following two rules:

$$\frac{}{even\_i\ 0}\ E0 \qquad \frac{even\_i\ n}{even\_i\ (S\ (S\ n))}\ E2$$

Rules *E0* and *E2* serve as canonical justifications for *even_i*, they are called the *constructors* of the inductive definition.

Now, assume a hypothesis H claiming that *even_i (S (S (S x)))* for some natural number x. Then, by looking at the definition of *even_i*, we see that only *E2* could justify H, and we can conclude that *even_i (S x)*. Similarly, *even_i* 1 can be considered as an absurd hypothesis, since *(S 0)* matches neither 0 nor *(S (S n))*, none of the two possible canonical ways of proving *even_i*, namely *E0* and *E2* can be used. Such proof steps are called *inversions*, because they use justifications such as *E0* and *E2* in the opposite way, i.e., from their conclusion to their premises. Note that *even_i* 3, *even_i* 5, etc. do not immediately yield the contradiction by inversion. However, by iterating the first inversion step, we eventually get *even_i* 1 and then the desired result using a last inversion. This illustrates that inversion is closer to case analysis than to induction.

Indeed, as we will see below, inversion can be decomposed into elementary proof steps, where the key step is a primitive case analysis on the considered inductive object (the hypothesis H, in our previous example). However, this decomposition is very often far from trivial because, in the general case, rules may include several premises, premises and conclusions may have several arguments and some of these arguments can be shared. Still, inversion turns out to be extremely useful in practice. Well-known instances are related to programming languages, whose semantics is described using complex inductively defined relations.

Note that it may be worth considering a (recursive) *function* for defining a predicate, rather than an inductive relation. For instance, in Coq syntax, an alternative way to specify even numbers is as follows:

[1] Co-inductive types are available as well. However, this paper does not depend on issues related to finiteness of computations: what is said about inductive types holds as well for co-inductive types.

```
Fixpoint even_f (n: nat) : Prop :=
  match n with
  | O ⇒ True
  | 1 ⇒ False
  | S (S n) ⇒ even_f n
  end.
```

Here *True* denotes a trivially provable proposition, and *False* denotes an absurd proposition. Using *even_f* is much simpler in the previous situations: for instance, *even_f (S (S (S x)))* just *reduces* to *even_f (S x)* using computation. In other words, computation provides inversion for free. Therefore, one may wonder why we should bother with inductively defined relations. Two kinds of answers can be given.

One of them is that an inductive definition allows us to focus exactly on the relevant values whereas, with functional definitions, we have to deal with the full domain, which can be much bigger in general. In our example above, suppose that we want to prove a statement such as $\forall n, even\ n \Rightarrow P\ n$. We can always attempt an induction on n, but this strategy forces to reason on all numbers, including odd numbers. If *even* is the recursive function above *even_f*, there is no other option. However, using *even_i*, we have the additional opportunity to make an induction on (the shape of) $even\_i\ n$, without needing to bother about odd numbers.

Another issue is that it is not always convenient or even possible to provide a functional definition of a predicate. Whenever possible, an n-ary relation R on $A_1 \times \ldots A_n$, is advantageously modeled by a function from $A_1, \ldots A_{n-1}$ to A_n. But it requires R to be functional (deterministic) and moreover, in type-theoretical settings such as CIC, to be total. If the relation is non-deterministic, we still can try to define it by a function returning either *True* or *False*, as is the case for *even_f*; this essentially amounts to providing a decision procedure for the intended predicate[2]. This is not always possible and, even if we can find such an algorithm, it may be hindered by undesired encoding tricks, which will induce additional complications in proofs. Moreover, a requirement of formal methods expresses that high-level definitions and statements should be as clear as possible in order to be convincing. The inductive style is not always better than the functional style, but it is often enough the case so that we cannot ignore it. For technical reasons, it is sometimes worth considering a functional version and an inductive version of the same notion. Even if the functional version is much better at inversion-like proof steps, the two versions have to be proved equivalent and there, the need for inverting the inductive version almost inevitably shows up.

Inductive relations are commonly used for defining the operational semantics of programming languages, either in small-step or in big-step style [11]. Such semantics define transitions between states, language constructs and, very often,

[2] Note that a 1-ary relation P on A_1 is isomorphic to a binary relation on $\mathbf{1} \times A_1$, where $\mathbf{1}$ is a type with exactly one inhabitant. If P holds for at least two values on A_1, it can be clearly considered as a non-deterministic function from $\mathbf{1}$ to A_1.

additional arguments such as input/output events. A tutorial example of a toy (but Turing-complete) language formally defined in Coq along these lines is given in [12] and routinely used as a teaching support in many universities. The Compcert project [6] is of course a much more involved example.

3 A Handcrafted Inversion

As noticed above, the heart of inversion is a suitable pattern matching on the hypothesis to be analyzed. With dependent types, it is possible for different branches to return a result whose type depends on the constructor. We make a systematic use of this feature: our key ingredient is a diagonalization function *diag*, which will be used for specifying the type returned on each branch. The exact shape of *diag* range from very simple to somewhat elaborated according to the goal at hand.

We recall the basics on dependent pattern matching, then we successively consider three situations, corresponding to increasingly complex variants of *diag*. In the two first situations, we consider inductive predicates with exactly one argument, for simplicity. The first situation is when all cases are absurd. The second is when a case is successful (or several cases) and we need to extract the information contained in successful cases, making new hypotheses in the environment. Then we show how to deal with additional arguments, so that constraints coming from the conclusion have to be propagated on the new hypotheses. Finally, we consider more elaborate dependent types and show how our technique works on a case where `inversion` fails.

3.1 Dependent Pattern Matching

To start with, let us take again the example of even numbers. Here is the corresponding Coq inductive definition.

```
Inductive even_i : nat → Prop :=
  | E0: even_i 0
  | E2: ∀ n, even_i n → even_i (S (S n)).
```

We see that each rule is given by a constructor in a dependent data type – also called an inductive predicate or relation because its sort is `Prop`. Therefore, the elementary way to decompose an object of type $even\_i\ n$ is to use dependent pattern matching. This is already done by primitive tactics of Coq such as `case` and `destruct`, which turn out to be powerful enough in many situations, when a condition is satisfied: the conclusion of the current goal fits all arguments of the hypothesis to be analyzed by pattern matching.

Let us first illustrate dependent pattern matching on even numbers. Consider a proof PE of type $even\_i\ n$ for some natural number n. For each possible constructor, *E0* or *E2*, we provide a proof term, respectively t_{E0} and t_{E2}. As usual, this term may depend on the arguments of the corresponding constructor, none for *E0* and, say x and ex for *E2*. More importantly for us, t_{E0} and t_{E2}

may have different *types*: the type $P\ n$ of the whole expression depends on n; in the first branch, the type of t_{E0} is $P\ 0$ and in the second branch, the type of t_{E2} is $P\ (S\ (S\ x))$. Therefore, the syntax of the `match` construct contains a `return` clause with the expected type of the result $P\ n$ as an argument; moreover, there is also an `in` clause for the type of PE which binds n:

```
match PE in even_i n return P n with
  | E0 ⇒ t_E0
  | E2 e ex ⇒ t_E2
end
```

Most of the time, Coq users do not need to go to this level of detail: in interactive proof mode, if n and $P\ n$ are clear from the context, `case` PE will do the job. More precisely, if we have an hypothesis H of type $even\_i\ n$ and a desired conclusion of type $P\ n$, `case` H will construct a proof term having the previous shape and answer with two new subgoals: one for $P\ 0$ and one for $P\ (S\ (S\ x))$, with $even\_i\ x$ as an additional assumption.

As a last remark, let us recall that an inductive type may have two kinds of arguments. We don't care about arguments which are "fixed" for all constructors: they are not even considered in pattern matching. In Coq they are called *parameters*. The other arguments are called *indexes*. For example, $even\_i$ has one index and no parameter.

3.2 Auxiliary Diagonalization Function

More work is needed precisely when there is no obvious relationship between the conclusion and the hypothesis to be analyzed. This happens in particular when H is absurd: the goal should be discharged whatever is its conclusion. This situation is covered as follows: the conclusion is converted to an expression $diag\ V$, where V is a value coming from H and $diag$ a suitable diagonal function, such that the dependent case analysis on H provides only trivial subgoals. For example, assume that we want to conclude $4 = 7$ from the hypothesis $H : even\_i\ 1$. Our diagonal function is then defined as follows.

```
diag x := match x with 1 ⇒ 4 = 7 | _ ⇒ True end
```

Then the conclusion is converted to $diag\ 1$, and the case analysis on H automatically provides two subgoals $diag\ 0$ and $diag\ (S\ (S\ y))$ for an arbitrary even natural number y. Each of these goals reduce to $True$, and we are done. The proof term behind this reasoning is very short (I is the standard proof of $True$):

```
match H in even_i n return diag n with E0 ⇒ I | E2 _ _ ⇒ I end
```

Such functions were already introduced in [9], but they work well only for handling absurd hypotheses. For instance, the examples presented below are out of reach of [9]. In order to explain how to extract information from satisfiable hypotheses, we start with an obvious generalization of the previous function for inverting absurd hypotheses.

3.3 Handling Successful Cases

A first easy improvement makes *diag* independent from the conclusion. To this effect, we replace it with $(\forall X, X)$ in the first branch of *diag*. In our previous example, this yields

$$diag\ x := \texttt{match}\ x\ \texttt{with}\ 1 \Rightarrow \forall\ (X\colon \texttt{Prop}),\ X \mid \_ \Rightarrow True\ \texttt{end}$$

Then the previous proof term (`match` H `in` $even\_i\ n$ `return` $diag\ n$ `with` 1 ...) has the type $\forall X, X$ and then can be successfully applied to any current conclusion. Alternatively, we can define a general function as follows:

```
Definition pr_1 {n} (en: even_i n) :=
  let diag x := match x with 1 ⇒ ∀ (X: Prop), X | _ ⇒ True end in
  match en in even_i n return diag n with E0 ⇒ I | _ ⇒ I end.
```

Next consider the following theorem:

$$\forall\ n\ m,\ even\_i\ n \to even\_i\ (n{+}m) \to even\_i\ m.$$

The proof is by induction on $even\_i\ n$. In the inductive step, we have to prove $even\_i\ m$ from the induction hypothesis $even\_i\,(n+m) \to even\_i\,m$ and a new hypothesis $H : even\_i\,(S\,(S\,(n+m)))$. Intuitively, we want to invert H in order to push $even\_i\,(n+m)$ in the environment. We can then adapt *pr_1* as follows:

```
Definition premises_E2 {n} (en: even_i n) :=
  let diag x :=
    match x with
      | S (S y) ⇒ ∀ (X: Prop), (even_i y → X) → X
      | _ ⇒ True
    end in
  match en in even_i n return diag n with
     | E2 p e ⇒ fun X k ⇒ k e
     | _ ⇒ I
  end.
```

Then, applying *premises_E2* to H yields a function in continuation passing style. Its type parameter X is automatically identified to the conclusion $even\_i\ m$, while y is bound to $n+m$, so that we get a new goal $even\_i(n+m) \to even\_i\,m$. That is, we have exactly the expected inversion. Functions such as *pr_1* and *premises_E2* can be seen as inversion lemmas, but note that their type is the dependent type expressed by their own *diag*.

More generally, let us then invert an hypothesis H having the type $A\,\mathcal{P}$ where $A(u)$ is an inductive type with index $u : U$ and $\mathcal{P} :\ U$ is an expression made of constructors in the type U. Given a constructor of type $\forall \mathbf{p}, A\ p$, where $\mathbf{p}$ is a telescope we proceed similarly: the `match` of *diag* has a first branch filtering $\mathcal{P}$ and returning $\forall X : Prop, (\forall \mathbf{p}, X) \to X$. If n constructors are possible for $A\,\mathcal{P}$, say respectively $C_1 : \forall \mathbf{p_1}, A\,\mathcal{P}$, ..., and $C_n : \forall \mathbf{p_n}, A\,\mathcal{P}$, the inverting lemma corresponding to $A\,\mathcal{P}$ will be:

```
Definition premises_Ap {u} (a: A u) :=
  let diag x :=
    match x with
      | P  ⇒ ∀ (X: Prop),  (∀ p1, X) → ... (∀ pn, X) → X
```

```
      | _ ⇒ True
    end in
  match a in A p return diag p with
    | C1 e1  ⇒ fun X k1... kn ⇒ k1 e1
⋮
    | Cn en  ⇒ fun X k1... kn ⇒ kn en
    | _ ⇒ I
  end.
```

Remark the close relationship with the impredicative encoding of data-types in system F.

3.4 Dealing with Constrained Arguments

The next stage to be considered is the case of an inductive type with more than one index. This raises new issues, because additional identities between arguments of the premises or the conclusion of a constructor may occur. This happens routinely in the inductive definitions for the operational semantics of C provided by CompCert. In order to explain the problems and how to deal with them in our framework, we introduce a toy language, together with an inductively defined evaluation rule *eval* having two indexes: the first one is the input type *tm*, *tm_const* and *tm_plus* are the expected cases in pattern matching; the second index is an output of type *val*, which is either nat or bool.

```
Inductive tm : Type :=
  | tm_const : nat → tm
  | tm_plus : tm → tm → tm.
Inductive val : Type :=
  | nval : nat → val
  | bval : bool → val.
Inductive eval : tm → val → Prop :=
  | E_Const : ∀ n,
      eval (tm_const n) (nval n)
  | E_Plus : ∀ t1 t2 n1 n2,
      eval t1 (nval n1) → eval t2 (nval n2) →
      eval (tm_plus t1 t2) (nval (plus n1 n2)).
```

In constructor E_Plus, the two premises share the variables $t1, t2, n1, n2$ with the conclusion. If we use the last solution with continuation passing style, as it is presented above, we are able to keep the premises but the relationship between the output values as specified in the inductive definition will be lost in the generated subgoal. This issue is handled using an additional argument to X corresponding to the second index of the inductive relation. The function for extracting the premises of E_Plus is:

```
Definition pr_plus_1 {t} {v} (e: eval t v) :=
  let diag t v :=
  match t with
    | tm_plus t1 t2 ⇒ ∀ (X:val → Prop),
```

```
      (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (nval (plus n1 n2)))
      → X v
  | _ ⇒ True
end in match e in (eval t v) return diag t v with
      | E_Plus _ _ n1 n2 H1 H2 ⇒ (fun X k ⇒ k n1 n2 H1 H2)
      | _ ⇒ I
    end.
```

Now, consider the following examples.

```
Lemma ex1: ∀ v, eval (tm_plus (tm_const 1) (tm_const 0)) v → v = nval 1.
Lemma ex2: ∀ n, eval (tm_plus (tm_const 1) (tm_const 0)) (nval n) → n = 1.
```

In *ex1*, by applying pr_plus_1, v will be equated to $nval\ (plus\ n1\ n2)$ according to the rule specified by E_plus. In *ex2*, we need to analyze at the same time the two arguments of *eval*. The corresponding premises are extracted using a function $pr\_plus\_1\_2$ having the same body as pr_plus_1, but whose type is:

```
match t, v with
  | tm_plus t1 t2, nval n ⇒ ∀ (X:nat → Prop),
    (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (plus n1 n2)) → X n
  | _, _ ⇒ True
end.
```

A similar situation happens with E_Const in the two previous examples.

Defining an inverting function for each constructor is most convenient for debugging. However the method is flexible and several such functions can be merged. In particular, an elegant alternative[3] is to provide a unique inverting function managing all cases of the argument(s) under focus. For instance, an exhaustive inverting function $pr\_eval\_1\_2$ suitable for *ex2* has the type:

```
match t, v with
  | tm_const c, nval n ⇒ ∀ (X:nat → Prop), X c → X n
  | tm_plus t1 t2, nval n ⇒ ∀ (X:nat → Prop),
    (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (plus n1 n2)) → X n
  | _, _ ⇒ ∀ X:Prop, X
end.
```

Full definitions as well as additional examples can be found on-line [10].

3.5 Beating `inversion`

Let us consider now a predicate defined on a dependent type. We take intervals $[1...n]$, formalized as t in the standard library `Fin`, then we restrict them to have an odd length.

```
Inductive t : nat → Set :=
    | F1 : ∀ {n}, t (S n)
    | FS : ∀ {n}, t n → t (S n).

Inductive odd : ∀ n : nat, t n → Prop :=
```

[3] We want to thank the anonymous reviewer who offered this remark.

```
  | odd_1 : ∀ n, odd (S n) F1
  | odd_SS : ∀ n i, odd n i → odd _ (FS (FS i)).
```

Finding the premises for the second constructor is a function similar to the one provided for $E2$ above:

```
Definition premises_odd_SS {n} {i: t n} (of: odd n i) :=
  let diag n i :=
    match i with
      | FS _ (FS _ y) ⇒ ∀ (X: Prop), (odd _ y → X) → X
      | _ ⇒ True
    end in
  match of in odd n i return diag n i with
      | odd_SS n i o ⇒ fun X k ⇒ k o
      | _ ⇒ I
  end.
```

In particular we can easily prove:

```
Lemma odd_SS_inv: ∀ n i, odd _ (FS (FS i)) → odd n i.
Proof. intros n i o. apply (premises_odd_SS o). trivial. Qed.
```

Standard `inversion` happens to fail here. Note that BasicElim may work (we actually could not succeed) but would need an additional axiom related to John Major equality.

4 Application to SimSoC-Cert

SimSoC-Cert [3,14] aims at certifying the simulator SimSoC, which is a complex hardware simulator written in C and C++. SimSoC is able to simulate various architectures including ARM and SH4 and is efficient enough to run Linux on them at a realistic speed. The main objective of SimSoC is to help designers of embedded systems: a large part of the design can be performed on software, which is much more convenient, flexible and less expensive than with real specific hardware components. However, this only makes sense if the simulator is actually faithful to the real hardware. Therefore we engaged in an effort to provide a formal certification of sensitive parts of SimSoC. More precisely, we consider the Instruction Set Simulator (ISS) for the ARM, which is at the heart of SimSoC. This ISS is called Simlight.

To this effect, first we defined a formal model in Coq of the ARM architecture, as defined in the reference manual [1]. Our second input is the operational semantics of the ISS encoded in C. This program is actually written in a large enough subset of C called Compcert-C, which is fully formalized in Coq [6].

We can then compare the behavior of the ISS encoded in C with the expected reference model directly defined in Coq. To this effect, a projection between the Coq model of the memory state of Simlight to the states in the reference model is defined. Then, correctness statements express that from a C memory m_1 corresponding to an abstract state s_1, performing the function claimed to represent a given instruction $\mathcal{I}$ in Simlight will result in a C memory m_2 which

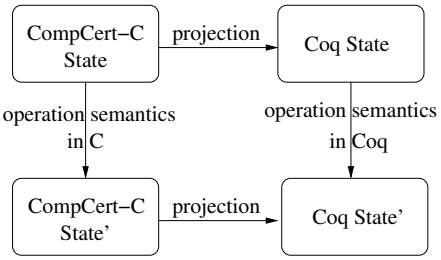

Fig. 1. Correctness of the simulation of an ARM operation

actually corresponds to the abstract state s_2 obtained by running the Coq model of $\mathcal{I}$. This can be put under the form of a commutative diagram as schematized in Fig. 1.

The operational semantics of C defining the evaluation is used everywhere in the proof: it provides the decomposition of the vertical arrow on the left column of Fig. 1 and drives the proof accordingly. We use the big step semantics, which is defined in CompCert by 5 mutually inductive transition relations. The largest inductive type for the evaluation of C expressions is *eval_expr*. It has 17 constructors, one for each CompCert C expression such as assignment, binary operation, dereference, etc.

In a typical proof step, we start from a goal containing a conclusion stating that a C memory state m_n and an ARM state st_n in the reference model are related by our projection, a hypothesis R_0 stating a similar relation between a C memory state m_0 and an ARM state st_0, and additional hypotheses $He_1, He_2, \ldots, He_n$. relating pairs of successive C memory states (m_0, m_1), $(m_1, m_2), \ldots, (m_{n-1}, m_n)$ respectively with (ASTs for) C expressions e_1, $e_2, \ldots,$ e_n, according to the relevant transition relation provided by CompCert. The general strategy is to propagate information from m_0 to m_1 using R_0 and He_1, then so on until m_n. To this effect we invert He_1, He_2, etc. However, according to the structure of e_1, inverting He_1 generates intermediate memory states and corresponding hypotheses that have to be inverted before going to He_2, unless e_1 is a base case. And sometimes, other kind of reasoning steps are needed, e.g., lemmas on the reference model of ARM.

For illustration, the following code shows a small excerpt from an old proof script in SimSoC-Cert using `inversion`. It corresponds to one line taken in an instruction called ADC (add with carry). It sets the CPSR (Current Program Status Register) with the value of SPSR (Saved Program Status Register). Lemma *same_cp_SR* states that the C memory state of the simulator and the corresponding formal representation of ARM processor state evolve consistently during this assignment. The pseudo-code from the ARM reference manual is just $CPSR = SPSR$. The corresponding C code is represented by the identifier *cp_SR* in the statement of the lemma.

```
Lemma same_cp_SR :
  ∀ e m l b s t m' v em,
    proc_state_related proc m e (Ok tt (mk_semstate l b s)) →
    eval_expression (Genv.globalenv prog_adc) e m cp_SR t m' v →
    proc_state_related proc m' e
      (Ok tt (mk_semstate l b
      (Arm6_State.set_cpsr s (Arm6_State.spsr s em)))).
```

After a couple of introductions and other administrative steps, we get the following goal, where *cp_SR* is unfolded in hypothesis H.

```
  ...
  l' : local
  b' : bool
  a' : expr
  H : eval_expr (Genv.globalenv prog_adc) e m RV
        (Ecall (Evalof (Evar copy_StatusRegister T14) T14)
          (Econs
            (Eaddrof
              (Efield (Ederef (Evalof (Evar proc T3) T3) T6)
                adc_compcert.cpsr T7) T8)
            (Econs
              (Ecall (Evalof (Evar spsr T15) T15)
                (Econs (Evalof (Evar proc T3) T3) Enil) T8) Enil))
          T12) t m' a'
============================
  proc_state_related m' e st'
```

Then we have to invert H and similar generated hypotheses until all constructors used in it type are exhausted. Here 18 consecutive inversions are needed. Using `inv`, which performs standard `inversion`, clearing the inverted hypothesis and rewriting of all auxiliary equations, the sequel of the script started as follows.

```
inv H. inv H4. inv H9. inv H5. inv H4. inv H5.
inv H15. inv H4. inv H5. inv H14. inv H4. inv H3.
inv H15. inv H5. inv H4. inv H5. inv H21. inv H13.
```

The names used there ($H4$, $H9$, etc.) are not under our control. The program for simulating an ARM instruction usually contains expression more complex than in the example given here. And unfortunately there is no clear way to share parts of the proofs involved since the corresponding programs are rather specific, at least for instructions belonging to different categories.

The drawbacks of the standard tactic `inversion` presented in the introduction show up immediatly. A first clue is the response time of Coq when inverting hypotheses H_i. Compiling the proof script corresponding to one instruction took more than a minute. About the naming issue, the constructors we face have up to 19 variables and 6 premises, yielding 25 names to provide. We could try to automate this naming using an ad-hoc wrapper around `inversion`, but things are complicated by the fact that this inversion program inserts additional hypotheses putting equational constraints between variables of the inverted constructor.

There are different ways to state and to place such constraints, and different releases of Coq may make different choices. The BasicElim approach introduces equations as well but from on our experiments, generated goals are much more regular than with `inversion`. In contrast, our approach does not suffer from such interferences, so we are anyway in a better position.

First, we define the diagonal-based function for each constructor of *eval_expr*, following the lines given in the previous section. For example, the evaluation of a field is defined in CompCert by the following rule.

```
Inductive eval_expr :
  env → mem → kind → expr → trace → mem → expr → Prop :=
  ...
  | eval_field: ∀ e m a t m' a' f ty,
      eval_expr e m RV a t m' a' →
      eval_expr e m LV (Efield a f ty) t m' (Efield a' f ty)
```

We then define (observe that 2 variables and 1 hypothesis will be generated):

```
Definition inv_field {g} {e} {m} {ex} {t} {m'} {ex'}
  (ee:eval_expr g e m LV ex t m' ex') :=
  let diag e ex ex' m m' :=
    match ex with
      | Efield a b c ⇒
        ∀(X:expr→Prop),
          (∀ t a', eval_expr g e m RV a t m' a' → X (Efield a' b c)) → X ex'
      | _ ⇒ True
    end in
    match ee in (eval_expr _ e m _ ex _ m' ex') return diag e ex ex' m m' with
      | eval_field _ _ _ t _ a' _ _ H1 ⇒ fun X k ⇒ k t a' H1
      | _ ⇒ I
    end.
```

Next we introduce a high-level tactic for each inductive type, gathering all the functions defined for its constructors. For example, *eval_expr* contains:

```
Ltac inv_eval_expr m m' :=
  ...
  let t1_:=fresh "t" in
  let v1_:=fresh "v" in
  let ev_ex1 := fresh "ev_ex" in
  ...
  match goal with
  ...
    | [ee: eval_expr ?ge ?e m LV (Efield ?a ?f ?ty) ?t m' ?a' ⊢ ?cl] ⇒
      apply (inv_field ee); clear ee; intros t1_ a1_ ev_ex1; intros;
      inv_eval_expr m m'
```

This tactic has two arguments m and m', corresponding to C memory states. The first `intros` introduces the 3 generated components with names respectively

prefixed by `t`, `v` and `ev_ex`. The second `intros` is related to previously reverted hypotheses, their names are correctly managed by Coq. Alltogether, such a tactic will:

1. Automatically find the hypothesis matching the arguments to be inverted;
2. Repeatedly perform our hand-crafted inversions for type *eval_expr* until all constraints between two memory states m and m' are derived;
3. Give meaningful names to the derived constraints;
4. Update all other related hypotheses according to the new variable names or values;
5. Clean up useless variables and hypotheses.

For example the 18 `inv` in the example above are solved in one step using *inv_eval_expr* m m', Note that the names are not explicitly given in the script, which would be cumbersome, but generated in our tactic.

Coq version changes had no impact on our scripts. Unexpectedly, changes in CompCert C semantics between versions 1.9 and 1.11 had no impact as well on proof scripts using our inversion. Of course, we still had to update the definition of diagonal functions.

Comparing development times provides additional hints. In our first try, using built-in inversion, more than two months were spent (by one person) on the development of the correctness proof of instruction ADC. Much time was actually wasted at maintaining the proofs since, as mentioned, a little change resulted in a complete revision of proof scripts. We then designed the inversion technique presented here. With the new approach, proofs for 4 other simple instructions could be finished in only one week, taking of course advantage of the previous experience with ADC. The high-level tactic described above required less than 2 weeks.

Finally, let us compare the efficiency of Coq built-in inversions (`inversion`, `derive inversion` which can generate an inversion principle once for all, and BasicElim [8]) with our inversion. We apply the four methods to the same examples, the lemma *cp_SR* and a single inversion on type *eval_expr* from CompCert C semantics. The first row is about the whole expression given in the example above. The other rows are inversions of specific expressions: *Ecall* is the CompCert-C expression of function calls, *Evalof* is to get the value of the specified location, *Eval* is to express constant, and *Evar* is to express variables. We can observe a gain of about 4 to 5 times. And generated object files are 5 times smaller.

Table 1. Time costs (in seconds)

	standard `inversion`	`derive inversion`	BasicElim	our inversion
Full example	1.628	0.976	1.428	0.312
Ecall	0.132	0.076	0.112	0.028
Evalof	0.132	0.072	0.092	0.020
Evar	0.128	0.064	0.084	0.024
Eaddrof	0.140	0.076	0.104	0.020

Table 2. Size of compilation results (in KBytes)

	standard `inversion`	`derive inversion`	BasicElim	our inversion
Full example	191	460	171	37

5 Conclusion

We see no reason why the technique developed above for performing inversions could not be automated and implemented in Coq or in proof assistants based on a similar calculus. One good motivation for that would be to get terms which are much smaller, easier to typecheck, than with the currently available inversion tactics. This can be very useful when interactively defining functions on dependent types, for instance.

But we want to insist first on a much more important feature of our approach, according to our experience with SimSoC-Cert: its impact on goals during *interactive* proof development is *actually controllable*. We think that having much shorter underlying functions is helpful in this respect: they are short enough to be written by hand, providing an exact view on what is to be generated. We claim that this feature is especially relevant to applications which make an intensive use of inversion steps: in this situation, partial automation obtained by programming small controllable building blocks turns out to be effective, whereas automation tends to generate a response of the proof assistant which is not completely predictible. This may not harm too much if the generated goals can be fully discharged without further interaction, but this is not the general case. In particular, this hope is vain when we deal with complex properties, as in our application. A better alternative would be to automatically generate auxiliary definitions such as *inv_field*. However, we consider that our technique is already useful and worth to be offered.

In contrast to available techniques [5,8] we argue against the use of auxiliary equations or disequations: the latter are better to be cleaned, in order to avoid clumsy additional hypotheses, which hamper the management of proof scripts; however, it is not that simple to do. The brute use of a tactic which performs all possible rewriting steps, then cleans equalities avalaible in the goal, for instance, is not satisfactory because some equalities already introduced by the user on purpose could then disappear. Therefore, a special machinery is needed in order to trace equalities coming from the inversion step under consideration (e.g., the use of *block* in BasicElim). Our use of CPS encoding of Leibniz equality, on the other hand, completely avoids this issue.

Our method was experimented on large proofs relying on big inductive relations independently defined in the Compcert project.

The current development can be found on-line [10], as well as examples given in Section 3.

Our group recently started another project dedicated to a certifying compiler from a high-level component-based language dedicated to embedded systems

(BIP), with CompCert C as its target. We expect the work presented here and our high-level tactics to be reused there.

Let us mention another possible application of the technique. Inversion is sometimes needed to write a function whose properties will be established later (as opposed to providing a monolithic and exhaustive Hoare-style specification and along with a VC generator such as Program). In this context simply using the proof engine and the `inversion` tactic tends to generate unmanageably large terms. We expect our technique to be very helpful in such situations.

References

1. ARM. ARM Architecture Reference Manual DDI 0100I. ARM (2005)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
3. Blanqui, F., Helmstetter, C., Joloboff, V., Monin, J.-F., Shi, X.: Designing a CPU model: from a pseudo-formal document to fast code. In: Proceedings of the 3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Heraklion, Greece (January 2011)
4. Chlipala, A.: Certified Programming with Dependent Types (2012), `http://adam.chlipala.net/cpdt`
5. Cornes, C., Terrasse, D.: Automating inversion of inductive predicates in coq. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 85–104. Springer, Heidelberg (1996)
6. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
7. McBride, C.: Inverting Inductively Defined Relations in LEGO. In: Giménez, E., Paulin-Mohring, C. (eds.) TYPES 1996. LNCS, vol. 1512, pp. 236–253. Springer, Heidelberg (1998)
8. McBride, C.: Elimination with a Motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 197–216. Springer, Heidelberg (2002)
9. Monin, J.-F.: Proof Trick: Small Inversions. In: Bertot, Y. (ed.) Second Coq Workshop, Royaume-Uni Edinburgh. Yves Bertot (July 2010)
10. Monin, J.-F., Shi, X.: Coq Examples for Handcrafted Inversions (2013), `http://www-verimag.imag.fr/~monin/Proof/hc_inversion/`
11. Nielson, H.R., Nielson, F.: Semantics with applications: A formal introduction. John Wiley & Sons, Inc., New York (1992)
12. Pierce, B.C., Casinghino, C., Greenberg, M.: Software Foundations (2009), `http://www.cis.upenn.edu/~bcpierce/sf`
13. Ricciotti, W.: Theoretical and Implementation Aspects in the Mechanization of the Metatheory of Programming Languages. PhD thesis, Università di Bologna (2011)
14. Shi, X., Monin, J.-F., Tuong, F., Blanqui, F.: First Steps Towards the Certification of an ARM Simulator Using Compcert. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 346–361. Springer, Heidelberg (2011)
15. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), `http://coq.inria.fr`